Bases de donnees: evaluation des programmes logiques recursifs par des fonctions recurrentes

Georges Gardarin, Christophe de Maindreville

IRIA
CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. : (1) 39 63 55 11

Rapports de Recherche

N° 473

BASES DE DONNÉES : ÉVALUATION DES PROGRAMMES LOGIQUES RÉCURSIFS PAR DES FONCTIONS RÉCURRENTES

Georges GARDARIN
Christophe de MAINDREVILLE

Janvier 1986

INRIA Internal Report, Nov. 1985

EVALUATION DES PROGRAMMES LOGIQUES RECURSIFS PAR DES FONCTIONS RECURRENTES

Georges GARDARIN
Christophe DE MAINDREVILLE
Projet SABRE
INRIA et UNIVERSITE PARIS VI (MASI)
BP 105, 78153 LE CHESNAY, FRANCE

RESUME

Dans un contexte de bases de données déductives, les auteurs proposent une nouvelle méthode pour compiler des questions référencant des relations définies par des prédicats récursifs. La méthode est basée sur une compréhension des questions et des prédicats comme des fonctions qui transforment des valeurs d'une colonne d'une relation en des valeurs d'une autre colonne. Il est montré qu'une large classe de questions avec des règles récursives associées peuvent être comprises comme des suites récurrentes de fonctions. Des cas typiques de suites récurrentes de fonctions sont résolues. Ces solutions conduisent à générer des programmes SQL optimisés ou à enchainer des opérateurs relationnels et des opérateurs de calcul de points fixes très efficaces. Des exemples d'applications sont présentés. En particulier, il est ainsi possible de résoudre trés efficacement toutes les questions invoquant des règles récursives linéaires acycliques.

EVALUATION OF DATABASE RECURSIVE LOGIC PROGRAMS AS RECURRENT FUNCTION SERIES

ABSTRACT

The authors introduce a new method to compile queries referencing recursively defined predicates. The method is based on an interpretation of the query and the relations as functions which map one column of a relation to another column. It is shown that a large class of queries with associated recursive rules (the tree query-rule class) can be computed as the limit of a serie of functions. Typical cases of series of functional equations are given and solved. The solutions lend themselves towards either extended relational algebra or SQL optimized programs to compute the recursive query answers. Examples of applications are given.

EVALUATION OF DATABASE RECURSIVE LOGIC PROGRAMS AS RECURRENT FUNCTION SERIES

Georges GARDARIN
Christophe DE MAINDREVILLE
SABRE Project
INRIA and UNIVERSITY PARIS VI
BP 105, 78153 LE CHESNAY, FRANCE

ABSTRACT

The authors introduce a new method to compile queries refencing recursively defined predicates. The method is based on an interpretation of the query and the relations as functions which map one column of a relation to another column. It is shown that a large class of queries with associated recursive rules (the tree query-rule class) can be computed as the limit of a serie of functions. Typical cases of series of functions are given and solved. The solutions lend themselves towards either extended relational algebra or SQL optimized programs to compute the recursive query answers. Examples of applications are given.

1. INTRODUCTION

Assuming the reader is familiar with deductive databases [Gallaire84], we address the problem of evaluating queries referencing recursively defined relations in a deductive database context. Several solutions have been proposed, among them [Chang81, Henschen84, Lozinskii85, Ullman85, Bancilhon85c].

Solutions have been classified in two types [Gallaire81] :

(i) Interpretation. This strategy is mainly derived from backward chaining and works in a top down manner. It suffers from two principal drawbacks. First, the halting and completness conditions of the deductive process are not easy to determine. They lead the approach to be rather difficult to implement [Vieille85]. Second, interpreted methods performs the inference basically one tuple at a time and does not take advantage of the efficient set manipulation primitives offered by relational DBMSs. However, interpreted methods are able to take advantage of the constants specified in the query when these constants propagate to a non recursive predicate in the recursive rules. Thus, they efficiently perform selections at first in favorable cases, although they inefficiently work one tuple at a time in straightforward implementations.

(ii) Compilation. This approach is mainly derived from forward

chaining and works in a bottom-up manner. The basic idea is to compile the query in a relational algebra program (including loops) which can be executed by the DBMS. A general and efficient compilation algorithm must generate programs which present the two nice following features [Bancilhon85a] :(a) No redundancy, that is tuples must not be produced twice using the same rules. The method described in [Bancilhon85b] is an example of such an improvement. (b) No generation of useless tuples, that is the constants which are given in the query must be used as soon as possible to eliminate tuples which are useless at the next steps of the derivation process. The method described in [Chang81] is an example of an efficient compilation method, but it only applies to regular rules. A uniform formalism called rule/goal graphs have been proposed to describe such methods [Ullman85]. Recently, the elegant notion of magic set have been introduced to avoid generation of useless facts [Bancilhon85c]; this approach applies to linear rules.

In this paper, we first propose a formalism which consists in interpreting relations and queries as functions. The intuitive idea is that a relation instance defines functions, each function being the mapping from one set of values in one column to the coresponding set of values in another column. Then, we use this functional formalism to translate queries and rules into recurrent series of functions. The resolution of the series appears then as a classical problem of mathematics. The approach applies for tree query-rules, that is for a large class of query-rules whose "connection graph" (see below) is a tree. The mehod is general in the sense that it works for regular, linear or non-linear tree query-rules. However, certain series are difficult to solve. When the resolution of a serie is feasible, the solution may be translated into either an optimized SQL program or in a program of relational algebra operators extended with fixpoint operators, such as transitive closure or extended transitive closure. Incidentally, we isolate a class of query-rules which may be translated into transitive closures. Also, we describe in details a SQL program generation method for linear rules. In conclusion, we introduce a new relational operator called external closure which permits to solve any linear tree query-rule, that is roughly speaking any acyclic linear rule.

The paper is organized as follows. First, we introduce some background on recursive Horn clause evaluation. This background is usefull in addition to [Gallaire84] and limited notions of graph theory for a good understanding of our method. Then, we introduce the functional view of relations and queries which is the basis of the method. In section four, we present the algorithm to translate recursive queries and the associated axioms into recurrent series of functions. Next, we present typical cases of functional equations which are easy to solve and to translate into SQL programs. This leads us to a clear distinction between regular rules, linear rules and more generally polynomial rules. The method allows us to solve most of the classical examples of recursive Horn clauses in the relational database context. Finally, we introduce the external closure operator which is going to be detailled in a forthcoming paper.

2. BACKGROUND ON RECURSIVE HORN CLAUSE THEORY

2.1 Logic Programs

A logic program consists of a set of Horn clauses. A set of Horn clauses can be used to derive a virtual relation from a relational databases. It has been shown that a set of possibly mutually recursive relations can be transformed into a set of Horn clauses with only one recursively defined predicate [Chandra85]. In this paper, we concentrate on logic programs which contains only one recursive predicate R. Such programs are composed of three sets of rules :

(i) A first set of rules allows the system to derive non recursive relations S0,S1...Sp from the base relations B0, B1 ...Bn; certain of the Si may be a base relation Bi; in that case the rule i is absent from this first set whose general form is as follows :

```
S1 <-- f1(B0,B1,...Bn)
S2 <-- f2(B0,B1,...Bn)
...
Sp <-- fp(B0,B1,...Bn)
```

(ii) A second set of rules gives an initial value to the recursive predicate R :

```
R  <-- g1(S1,S2,...Sp)
R  <-- g2(S1,S2,...Sp)
...
R  <-- gq(S1,S2,...Sp)
```

(iii) Finally, the third set allows the system to compute recursively R :

```
R  <-- h1(R,S1,S2,...Sp)
R  <-- h2(R,S1,S2,...Sp)
...
R  <-- hr(R,S1,S2,...Sp)
```

In all the above rules, fi, gi and hi are functions of the given argument relations (some may be absent). For simplicity, we assume that all arguments of our predicates are variables; thus, fi, gi, and hi are relational algebra expressions composed with joins and projections. Every logic program can be cast into the given form by eliminating mutual recursion and by representing the constants as facts.

It is known [Chandra82] that the recursive relation which derives from the given set of Horn clauses is the least fixpoint of the equation :

```
R = g1(S1,S2,...Sp) U g2(S1,S2,...Sp) U ... U gq(S1,S2,...Sp) U
    h1(R,S1,S2,...Sp) U h2(R,S1,S2,...Sp) U ...U hr(R,S1,S2,...Sp)
```

which can be rewritten simply as :

```
R = G(S1,S2...Sp) U H(R,S1,S2,...Sp)
```

or equivalently as:

$$R = F(R)$$

where, G, H and F are composed with joins, projections and unions of relations chosen among R,S1,S2 ... Sp.

2.2 Least fixpoint operators

Let us consider a relation R defined recursively by an equation of the form :

$$R = F(R) \qquad (1)$$

where F(R) is a relational expression with operand R, perhaps among other operands S1,S2 ...Sp. F is a function over the relations having the same schema as R. Such relations form a complete lattice using the set inclusion. Thus, if F is increasing (i.e. R1 >= R2 implies F(R1) >= F(R2)) , the Tarski theorem [Tarski55] applies and we know that equation (1) has a set of solutions which is a complete lattice. Moreover, as all argument relations are finite, it has been demonstrated [Aho79] (the demonstration is simply done by induction on i using the finitude hypothesis) that : (0 denotes the relation with no tuple) :

(i) The sequence F(0), F(F(0)),... $F^n(0)$ is convergent towards a term denoted lim $F^n(0)$.

(ii) Lim $F^n(0)$ is the least fixpoint of equation (1).

It is important to point out that every relational expression composed with union, join and projection is increasing [Aho79]. Thus, the previous result directly applies to a recursive relation defined by a logic program.

This result leads to a very simple way to compute a recursive relation R as the least fixpoint of equation (1). This can be done using the naive program [Bancilhon85a]:

```
R := F(0);
while R changes  do  R := F(R);
```

An elegant optimization of this procedure using a differential approach can be found in [Bancilhon85b]. Nevertheless, such a procedure may be inefficient to solve queries because useless tuples may be generated. The difficult problem is to perform the possible selections derived from the query before applying the fixpoint computation algorithm. In the sequel, we propose a general solution to this problem.

3. RELATIONS AND QUERIES AS FUNCTIONS

3.1 Functions defined by a relation

Let R(A1,A2,...An) be a relation of attributes A1,A2...An. Let i and j be two subsets of (1,2,...n). We shall use the notation Ai (resp. Aj) to denote the set of attributes indexed by the elements of i (resp. j). For example, with i = (2,4), Ai will designate A2,A4. In most cases, i and j will be subsets of one indice; therefore, i (resp. j) may be seen as the rank of attribute Ai (resp Aj) in the relation R. We denote dom(i) (resp. dom(j)) the domain of Ai (resp. Aj), that is in general the cartesian product of the domains of the composing attributes. A given instance of R

determines a function R:i->j defined as follows.

Definition 1: Relational function

> A relational function R:i->j is a function such that :
> (i) The definition domain is the power set of dom(Ai), and the image domain is a subset of the power set of dom(Aj).
> (ii) R:i->j(x) where x = {x1,x2,...xp} is a subset of dom(Ai), is the subset y = {y1,y2,...yq} of dom(Aj) obtained by restricting R to those tuples having x1 or x2 or ...xn for value of Ai, keeping only the values of Aj as a set:
> $y = set(\Pi_{Aj}(\sigma_{Ai=x1 \text{ OR } ... \text{ OR } Ai=xn}(R)))$.

In other words, y is all the values yq for which there exists a tuple in R with values xp for attribute(s) Ai and yq for Aj, where xi belongs to x; if none, y is the empty set. Intuitively, R:i->j is the function which leads from a set of values of attributes Ai to the corresponding set of values of attributes Aj following the relation R.

Let us give examples of relational functions derived from the relation instance PARENT portrayed figure 1.

PARENT	PARENT	CHILDREN	BIRTH_DATE
	lulu	toto	1970
	tintin	lulu	1945
	lili	toto	1970
	titine	lulu	1945

Figure 1 : An instance of the PARENT relation

Thus, the previous relation defines the following functions :

(i) PARENT:1->2(x) which determines the children of given parents; for example, we get :
PARENT:1->2(0) = 0 where 0 is the empty set;
PARENT:1->2({lulu}) = {toto};
PARENT:1->2({lulu,tintin}) = {toto,lulu};
PARENT:1->2({lulu,tintin,totoche}) = {toto,lulu};
PARENT:1->2({totoche}) = 0.

(ii) PARENT:2->1(x) which determines the parents of given children; for example, we get:
PARENT:2->1(0) = 0;
PARENT:2->1({toto}) = {lulu,lili};
PARENT:2->1({toto,lulu}) = {lulu,tintin,lili,titine};
PARENT:2->1({toto,totoche} = {lulu,lili}.

(iii) PARENT:1->2,3(x) which determines the children with birthdate for given parents; for example, we obtain :
PARENT:1->2,3({lulu}) = {toto-1970};
PARENT:1->2,3({lulu,tintin}) = {toto-1970,lulu-1945};

(iv) PARENT:2,3->1(x) which determines the parents of given

children with specified birthdates; for example, we have :
PARENT:2,3->1({toto-1945}) = 0;
PARENT:2,3->1({toto-1970}) = {lulu,lili}.

3.2 Relational function sum and composition

We shall use the following classical operations over functions and function results :
(i) The union of two sets resulting from a function evaluation is denoted +. We obviously have f(x+y) = f(x) + f(y).
(ii) The sum of two function having same domain is defined by:
$(f+g)(x) = f(x) + g(x)$.
(iii) The composition of f and g is possible if the image domain of g is included in the definition domain of f; it is defined by :
$f \circ g\ (x) = f(g(x))$.

3.3 Selections as function evaluations

In this section, we consider possible queries on a relation using the equal (=) comparator and the logical operators "or" and "and". We shortly show that such queries can be expressed simply as a function evaluation.

Lemma 1:

> Any selection request over a relation R can be represented as a sum of relational function evaluation.

Proof:

A selection of a relation R can be expressed using relational algebra as $\Pi_{Aj}(\sigma_{Q}(R))$. The result being a set of Aj values, the image domain of the functions which may be included in the sum is $2^{dom(Aj)}$. Any disjunctive restriction on attribute(s) Ai of a relation R(A1,A2...An) followed by a projection on attribute(s) Aj can be expressed as a function evaluation as follows:
$\Pi_{Aj}(\sigma_{Ai=c1 \text{ or } \ldots \text{ or } Ai=ck}(R)) = R:i->j(\{c1,\ldots ck\})$
Any conjunctive restriction on attributes Ai1,Ai2,...Aik (with Aip different from Aiq for all p,q otherwise the result is empty and the empty function may be used) followed by a projection on attribute(s) Aj can be expressed as a function evaluation as follows :
$\Pi_{Aj}(\sigma_{Ai1=c1 \text{ and} \ldots Aik=ck}(R)) = R:i1,\ldots ik->j(\{c1,\ldots ck\})$
More generally, it is always possible to put the qualification Q in disjunctive normal form. Each conjunction may then be replaced by a relational function evaluation. The total query is then equivalent to the sum of all the relational function evaluations derived from each conjunction. □

Let us give some examples of simple selections expressed as relational function evaluation using the PARENT relation :
(i) Give the parents of toto :
PARENT:2->1({toto}).
(ii) Give the parents of toto or lulu :
PARENT:2->1({toto,lulu}).
(iii) When does lulu become the parent of toto ?

PARENT:1,2->3({lulu-toto}).

3.4 Semi-join as function composition

It is important to point out that the composition of two functions f and g derived from two relations R and S may be defined using a relation which is a join of R and S. More precisely, we can demonstrate the following lemma which shows that a function composition is really equivalent to a semi-join [Bernstein79].

Lemma 2

> Any sequence of the following relational algebra operations:
> (i) selection of a relation R on attribute(s) Ai,
> (ii) natural join of the result with relation S,
> (iii) projection of the result on attribute(s) Bq,
> can be represented as a composition of two relational functions derived from R and S.

Proof

Let R(A1,A2,...An) and S(B1,B2, ...Bm) be two relations. Let:

$$E = \overline{\Pi}_{Bq} \left(\overline{\sigma}_{Ai=x}(R) \underset{Aj=Bp}{\bowtie} S \right)$$

Then, using the definition of the relational functions, we can demonstrate obviously :

$$E = S{:}p\text{->}q \circ R{:}i\text{->}j\ (x). \quad \square$$

A corollary of this lemma that will be used in the demonstration of the next lemma of this paper the following.

Lemma 3 :

> The composition of two relational functions f o g can be expressed as a unique relational function using the join of the two relations defining the relational functions f and g.

Proof

Let $E = S{:}p\text{->}q \circ R{:}i\text{->}j\ (x)$ and let $T = R \underset{Aj=Bp}{\bowtie} S$.

We can show using the definition of the relational functions :

$$E = T{:}i\text{->}q(x). \quad \square$$

4. EVALUATION OF RECURSIVE QUERIES

4.1 Recursive queries

A recursive query is a query on a recursively defined relation R. Using section 3, it appears that the evaluation of a recursive query expressed in SQL as follows :

SELECT Aj FROM R WHERE Ai = c

consists in evaluating the function:

R:i->j(c).

Thus, recursive query evaluation leads to relational function evaluations. For simplicity, we shall restrict ourself to the case

where Aj and Ai are unique attributes. Generalization is feasible.

4.2 Recursive relations as series of functions

Let R be a recursively defined relation. As stated in section 2, R is the least fixpoint of an equation R = F(R) where F is a relational algebra expression. Using the notations R0=0 , R1=F(0), ...,Rn-1=F^{n-1}(0), Rn=F^{n}(0), the solution of the equation is R = lim Rn and we have Rn=F(Rn-1). In the sequel, we are going to show that in most cases, any function Rn:i->j (be fn) can be expressed as an expression of the function Rn-1:i->j (be fn-1) possibly combined with other known functions, where the expression is composed with the function addition (+) and the function composition (o). For example, for a given query referencing a predicate defined with linear recursive rules , we shall generally obtain a serie of function of the form fn(x) = G(x) + H o fn-1 o K(x) with f0(x) = 0. Such a serie is rather easy to solve with traditional mathematics. In the case of non linear rules, the serie is just a little bit more complex.

Let us justify the methodology we are going to use which consists in studying the recursive rules one by one and to collect the results to generate the recurrent serie of function Rn:i->j(x). As shown above, a query on R requires the evaluation of a relational function R:i->j. Using the equation Rn = F(Rn-1), we obtain :

Rn:i->j(x) = F(Rn-1):i->j(x) for all x (2).

Let us recall from section 2 that F is a union of relational algebra expressions derived from each rule having R as left predicate. More formally, using the notation of section 2, we have :

F(R) = g1(S1,S2,...Sp) U g2(S1,S2,...Sp) U ... U gq(S1,S2,...Sp) U
h1(R,S1,S2,...Sp) U h2(R,S1,S2,...Sp) U ... U hr(R,S1,S2,...Sp)

Therefore R:i->j(x) appears to be the limit of the serie of functions Rn:i->j(x) defined by :

Rn:i->j(x) = g1(S1,...Sp):i->j(x) + g2(S1,...Sp):i->j(x) ...+ ...
+ h1(Rn-1,S1,...Sp):i->j(x) + h2(Rn-1,S1,...Sp):i->j(x) + ... (3)

In the above formula, it appears that the g1, g2 ... functions derive from relations which can be computed without recursion from the base relations. It is not the case for the h1, h2 ... functions. However, we are going to show that in most cases hi(Rn-1,S1,...Sp):i->j(x) can be expressed as Hi(Rn-1:i->j)(x) where Hi is an expression composed with the classical function composition o. In addition, formula (3) shows formally that the axioms of the logic program can be studied one by one. The study of each of the non recursive axioms leads to a formula of the following form :

Rn:i->j(x)$\supset$ gi(S1,...Sp):i->j(x)

while the study of each of the recursive axioms leads to a formula of the following form :

Rn:i->j(x)$\supset$ Hi(Rn-1:i->j)(x).

Formula (3) shows that after studying each axiom, we can collect the results to get the serie definition :

R0:i->j(x) = 0;

Rn:i->j(x) = g1(S1,...Sp):i->j(x) + ... + gi(S1,...Sp):i->j(x) +

...+ H1(Rn-1:i->j)(x) + ... + Hi(Rn-1:i->j)(x) +...

In the next section, we give an algorithm to derive Hi(Rn-1:i->j) from the recursive axiom R <- hi(R,S1,S2...Sp). Hi(Rn-1:i->j) is called above the contribution of the rule to the query.

4.3 Connection graph of a rule

To translate a query with the associated rules into a functional equation, we may use a connection graph for each rule. The connection graph of a rule represents all the possible functional expressions which may be derived from a rule. Such a graph is more precisely defined as follows.

Definition 2: Rule connection graph

> A connection graph of a rule is an oriented labelled graph such that:
> (i) A node corresponds to each variable appearing in the the rule.
> (ii) For any pair of variables x and y appearing in the same predicate Q, there exists two arcs x->y and y->x labelled with the corresponding relational functions Q:i->j and Q:j->i, indiced with n-1 if it is a right occurrence of the recursive predicate and with n if it is a left occurrence of the recursive predicate.

It is important to point out that in this graph, any arc represents a function whose definition is given by the arc label, while a node portrays a variable. Figure 2 gives an example of a connection graph for the rule :

A(x,y) <- A(x,z) ^ P(z,y)

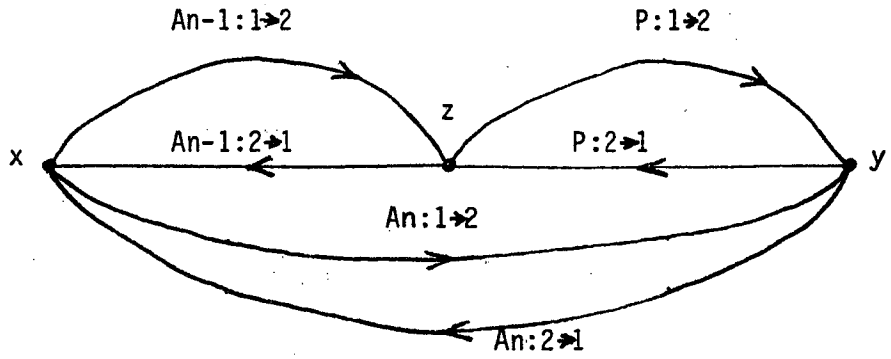

Figure 2 : Example of a rule connection graph.

To derive the functional expression, we more precisely use a query-rule connection graph corresponding to a given query and a given rule. Such a graph represents the usefull functional expressions for solving the query. It is defined as follows.

Definition 3: query-rule connection graph

A query-rule connection graph of a rule R <- h(R,S1, ...,Sp) with regards to a query R:i->j(c) is obtained from the rule connection graph as follows:
(i) Mark the origin of the arc with label Rn:i->j as the root and its extremity as the target;
(ii) Leave out all arcs with label Rn:p->q and Rn-1:p->q whatever be p and q, except the arcs with label Rn-1:i->j.

Clearly, the query-rule connection graph keeps the arcs representing the recursive predicate of type Rn-1:i->j, that is the functions which correspond to the query to evaluate. Let us point out that no nodes are suppressed when going from the rule connection graph to the query-rule connection graph. The query-rule connection graph is in a certain sense the "filtering" of the rule connection graph by the query. Figure 3 gives an example of a query-rule connection graph for the rule :

$$A(x,y) \leftarrow A(x,z) \wedge R(z,y)$$

and the query :

$$y = A:1\rightarrow 2(c).$$

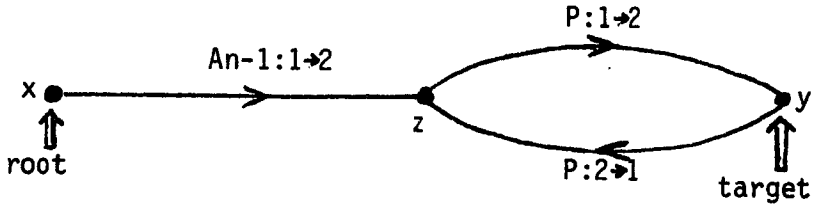

Figure 3: Example of a query-rule connection graph.

4.4 Translation of a recursive rule into a function composition

Indeed, a very general class of recursive rules and queries leads to series of recurent functions as shown below. This class is called tree query-rule and defined as follows.

Definition 4: Tree query-rule

A tree query-rule is a couple query-rule whose query rule connection graph satisfies the following properties :
(i) The query-rule connection graph can be transformed in an oriented tree from the root to the leaves by choosing one arc when two nodes are linked by two arcs not labelled by the recursive predicate.
(ii) The recursive predicate appears only in the path going from the root to the target.

In practice, most of the meaningfull examples of recursive rules seem to lead to tree query-rules as defined above. We would be able to extend slightly the notion of tree query-rule to include query-rule connection graph with no cycle of more than two nodes. This will uselessly complicate the following demonstrations which show that tree query-rules may be solved as recurrent functions.

Lemma 4 :

> The contribution of a rule R<-h(R,S1,...Sp) to a query R:i->j(c) can be expressed as a composition of functions derived from S1, ...Sp with the function Rn-1:i->j if the couple query-rule is a tree query-rule.

Proof :

The proof consists in giving an algorithm to derive the functional composition from the tree derived from the query-rule connection graph. Since the oriented query-rule graph may be seen as a tree of root x and target y, there exists a unique path in that tree going from x to y. Let (x,z1,G1), (z2,z3,G2), ... (zn,y,Gn) be this oriented path from x to y, where zi are nodes and Gi is a relational function which labels the arc (zi,zi+1). The functional composition is obtained by the following transformation algorithm :

(i) Replace each branch leaving out the previous path by a unique arc (zi,u,H:r->s) where H is the correct function derived from the join of all the relations which correspond to the labels of the branch arcs; this is possible because of lemma 3; due to the second part of the definition of a tree query-rule, this join does not involve the recursive predicate and is computable using S1,...Sp.

(ii) If there exists one arc (zi,u,H:r->s) where u is not on the path from x to y , then delete this arc and replace the arc (zi-1,zi,Gi) by (zi-1,zi,H:r->r o Gi).

(iii) Let (x,z1,H1), (z2,z3,H2), ...(zn,y,Hn) be the remaining path going from x to y. The functional expression is simply :

$$Hn \circ Hn\text{-}1 \circ \ldots H1\ (x).$$

Indeed, this expression is a composition of Rn-1:i->j with other functions derived from the base relations. □

4.5 Examples

We now present few examples of queries on recursive relations. For each example, we give the transformation of the query in the evaluation of a serie of functions.

4.5.1 The ancestors

Let PARENT(CHILDREN,PARENT) be a base relation that we refer as P. The ANCESTOR relation referred as A can be defined as follows:

A(x,y) <- P(x,y) (1)
A(x,y) <- A(x,z) ^ P(z,y) (2)

Let us assume the query :
SELECT PARENT FROM ANCESTOR WHERE CHILDREN = toto
or, with logical notations : ANCESTOR(toto,?y).
Such a query requires to evaluate the function :
A:1->2(x) for x={toto}.
Clause (1) gives obviously :
An:1->2(x) $\supset$ P:1->2(x) (1').

The query-rule connection graph of clause (2) is portrayed figure 3. Applying the transformation algorithm, we obtain :

An:1->2(x) ⊃ P:1->2 o An-1:1->2(x) (2').

Collecting (1') and (2'), the recurent definition of the ancestor function turns out to be :

An:1->2(x) = P:1->2(x) + P:1->2 o An-1:1->2(x),

with :

A0:1->2(x) = 0.

If we simplify the notations by leaving out the function definition domains (1->2) which are identical, we have :

A0(x) = 0

An(x) = P(x) + P(An-1(x))

which is rather simple to solve using classical mathematics, as shown below.

4.5.2 The friend of the family

Let FRIEND(PERSON1,PERSON2) be a base relation denoted F. Using the other base relation PARENT(CHILDREN,PARENT), we can derive the FAMILY_FRIEND relation denoted FF as follows:

FF(x,y) <- F(x,y) (1)

FF(x,y) <- P(x,z) ^ FF(z,y) (2)

Let us assume that we want to find the family friends of toto, that is the answer to the query FF(toto,?y). For this purpose, it is necessary to evaluate the function FF:1->2(x) where x = {toto}.

Clause (1) gives obviously :

FFn:1->2(x) ⊃ F:1->2(x) (1')

The query-rule connection graph derived from clause (2) is portrayed figure 4.

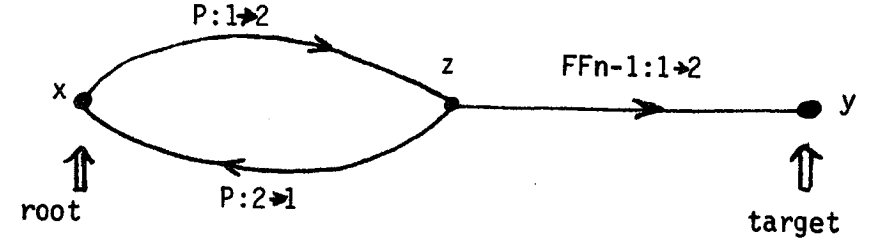

Figure 4 :Query-rule connection graph for the family friends.

Using the transformation algorithm, we obtain :

FFn:1->2(x) ⊃ FFn-1:1->2 o P:1->2 (x) (2')

Collecting (1') and (2'), the recusive definition of FFn:1->2 turns out to be :

FFn:1->2(x) = F:1->2(x) + FFn-1:1->2 o P:1->2 (x)

with :

FFo:1->2(x) = 0.

To clarify for the reader, we can use the simplified notations :

FFo(x) = 0;

FFn(x) = F(x) + FFn-1(P(x)).

This is a little bit more complexe to solve that the previous example, although not to complex as we shall see below.

4.5.3 The cousin of the same generation

Let us assume the base relation PARENT(CHILDREN,PARENT) and a derived relation HUMAN(NAME) which can be derived from the PARENT relation by the following non recursive axioms :

```
HUMAN(x) <- PARENT(x,y)
HUMAN(y) <- PARENT(x,y)
```

The purpose of the example is to compute the recursive predicate cousin of same generation (SG) [Bancilhon85c] defined as follows from the HUMAN (H) and PARENT (P) predicates :

```
SG(x,x)  <- H(x)                                  (1)
SG(x,y)  <- P(x,xp) ^ SG(xp,yp) ^ P(y,yp)         (2)
```

The considered query consists in finding the same generation cousin of "Agamemnon", that is : SG(Agamemnon,?y).
This leads us to evaluate the function :
SG:1->2(x) for x = {Agamemnon}.

Obviously, (1) gives :
SGn:1->2(x) ⊃ H:1->1(x) (1')
Let us point out that H:1->1(x) is the identity function and could be denoted I(x).
The query-rule connection graph for the rule (2) is portrayed figure 5.

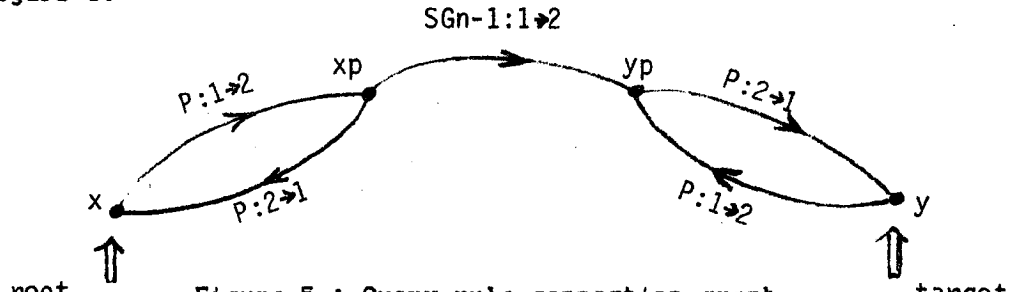

Figure 5 : Query-rule connection graph for the same generation cousins.

The application of the transformation algorithm yields :
SGn:1->2(x) ⊃ P:2->1 o SGn-1:1->2 o P:1->2 (x) (2')
Collecting (1') and (2'), we finally obtain the recurent system :

```
SGO:1->2(x) = 0
SGn:1->2(x) = H:1->1(x) + P:2->1 o SGn-1:1->2 o P:1->2 (x)
```

which can be simplified as follows (be carefull, that both P:2->1 and P:1->2 must be used; we call them P' and P below) :

```
SGO(x) = 0
SGn(x) = I(x) + P'(SGn-1(P(x))).
```

4.5.4 The ancestors of even generations

To illustrate a non linear rule, we define the ancestor of even generations (AA) from the parent relation (P) as follows :

AA(x,y) <- P(x,z) ^ P(z,y) (1)
AA(x,y) <- AA(x,z) ^ AA(z,y) (2)

Another possible non linear example is the ancestors with a definition using ancestor and ancestor. Using the even generation ancestor, we consider the query AA(toto,?y) which gives the even ancestors of toto.

To solve the previous query, we need to evaluate the function: AA:1->2(x) where x = {toto}.
From rule (1), we derive :
AAn:1->2(x) ⊇ P:1->2 o P:1->2 (x) (1')
The query-rule connection graph of rule (2) is portrayed figure 6.

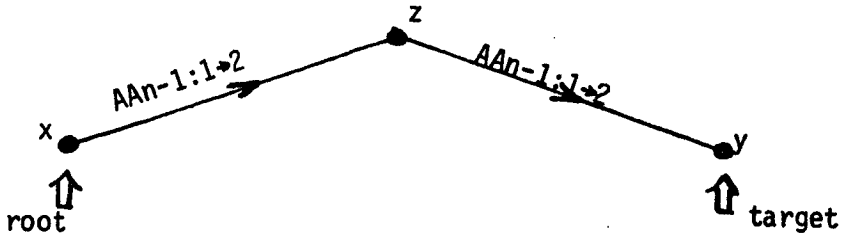

Figure 6 : Query-rule connection graph for the even ancestors.

We derive from this graph the equation :
AAn:1->2(x) ⊇ AAn-1:1->2 o AAn-1:1->2 (x) (2')
Collecting (1') and (2'), we get :
AAn:1->2(x) = P:1->2 o P:1->2 (x) + AAn-1:1->2 o AAn-1:1->2 (x)
Simplyfing the notation, we have to solve the recurrent functions definition system :

AA0(x) = 0
AAn(x) = P(P(x)) + AAn-1(AAn-1(x))

5. SOLVING RECURRENT SERIES OF FUNCTIONS

In this section, we do not intend to give a course on solving recurent series. This is a problem of mathematics which is rather well known. We consider some interesting cases below. From a typologie of the series which result from the rules, we propose to classify the system of rules.

5.1 Regular systems

Definition 5: Regular systems
A regular system is a set of recursive rules which leads to a regular serie of the form :

R0(x) = 0
Rn(x) = S(x) + Q(Rn-1(x))
where S and Q are functions derived from base relations or from non recursive expressions of base relations.

The given type of series is obtained from a particular case of linear clauses, when the unique path determined by the query in the query-rule connection graph starts with the recursive predicate. Such rules are those which leads to regular grammars in the Chang approach [Chang81]. An example of such a serie is the ancestor example.

Regular systems are easy to solve. For this purpose, we shall use the notation Q^n to represent the composition of Q o Q o ...o Q (n times). Starting with n = 1, we have :

$R1(x) = S(x)$
$R2(x) = S(x) + Q(R1(x)) = S(x) + Q(S(x))$
$R3(x) = S(x) + Q(R2(x)) = S(x) + Q(S(x)) + Q^2(S(x))$

To proceed by induction on n , let us assume that :
$Rn-1(x) = S(x)+Q(S(x))+Q^2(S(x))+...+Q^{n-2}(S(x))$

Then, applying the formula Rn(x) = S(x) + Q(Rn-1(x)), we have :
$Rn(x) = S(x)+Q(S(x)+Q(S(x))+...+Q^{n-2}(S(x)))$
which is obviously, as Q is an additive function :
$Rn(x) = S(x)+Q(S(x))+Q^2(S(x))+...+Q^{n-1}(S(x))$

Thus, the induction hypothesis being true, we can assert for all n:

$$Rn(x) = S(x)+Q(S(x))+Q^2(S(x))+...+Q^{n-1}(S(x))$$

Before generating a program for solving a regular system, it is interesting to remark that a regular system is not computable as a transitive closure of a relation. In a slightly more general way as usual, we call transitive closure of a relation the union of the projections on the relation schema of all the successive joins of the relation with itself on chosen attributes, using the relation 1,2,3...n,... times. Indeed, only a sub-case of regular systems is computable with a transitive closure operator as stated in lemma 5.

Lemma 5

Regular systems of Horn clauses which lead to regular series of the form:
R0(x) = 0
Rn(x) = S(x) + S(Rn-1(x))
where S is a function derived from base relations or from non recursive expressions of base relations are computable with a transitive closure operator.

Proof:
Using the solution for Rn(x) and replacing Q by S, we obtain:

$$Rn(x) = S(x) + S^2(x) + \ldots + S^n(x)$$

The limit of Rn is obviously the transitive closure of S. □

Let us now generate a program to compute R(x) for regular systems. Assuming that Q^k is different from Q^n for all n,k with n different of k (i.e. that Q is a non cyclic function), the Rn(x) solution formula can be easily transform in an iterative program to compute R(x) which is the limit of Rn(x) when n grows. This program is given below :

```
Procedure evaluate(R,x);
  Begin
    R := 0;
    DR := S(x);
    while DR <> 0 do
      R := R + DR;
      DR := Q(DR);
    od;
  end;
```

The above procedure is correct for any recursive predicate R defines by regular clauses (i.e., clauses which leads to regular series). Let us point out that x can be any set of values. The addition is the set union and the function composition is the one defined above.

The previous result can be applied to the ancestor example as the parent and ancestor relations are acyclic. The serie obtained was a regular one as defined above where :

Rn is called An;
S is called P;
Q is called P.

Therefore, the procedure to compute A(x) is :

```
Procedure evaluate(A,x);
  Begin
    A := 0;
    DA := P(x);
    while DA <> 0 do
      A := A + DA;
      DA := P(DA);
    od;
  end;
```

As P(x) is indeed P:1->2(x), we can move back to relational algebra using SQL notations. Thus, we get the following procedure to evaluate the ancestors of toto :

```
Procedure evaluate(A,toto);
  Begin
    A := 0;
    DA := SELECT PARENT FROM PARENT WHERE
          CHILDREN=toto;
    while DA <> 0  do
      A := A U DA;
```

```
        DA := SELECT PARENT FROM PARENT WHERE
              CHILDREN IN DA;
    od;
end;
```

5.2 Linear systems

Definition 6: Linear system

A linear system is a set of recursive rules which leads to a linear serie of the form :

$R0(x) = 0$

$Rn(x) = S(x) + Q(Rn-1(T(x))$

where S, Q and T are functions derived from base relations or from non recursive expressions of base relations.

Linear systems are more general than regular systems, as regular series are the linear series in which T is the identity. Linear series are obtained from linear clauses, when there exists a unique path in the query-rule connection graph as seen above. Such rules are most of the linear rules. Examples of such series are the family friends and the same generation cousins examples.

Linear series may be solved using a similar process as the one used for regular series. Indeed, we have :

$R1(x) = S(x)$

$R2(x) = S(x) + Q(S(T(x)))$

$R3(x) = S(x) + Q(R2(T(x))) = S(x)+Q(S(T(x))+Q^2(S(T^2(x)))$

Let us assume as induction hypothesis :

$Rn-1(x) =$
$S(x)+Q(S(T(x)))+Q^2(S(T^2(x)))+...+Q^{n-2}(S(T^{n-2}(x)))$

Then, we get :
$Rn(x) = S(x) + Q(S(T(x))+Q(S(T(T(x))))+ Q^2(S(T^2(T(x))))+$
$...+Q^{n-2}(S(T^{n-2}(T(x)))))$
and, as Q is additive, we get :
$Rn(x) =$
$S(x)+Q(S(T(x)))+Q^2(S(T^2(x)))+...+Q^{n-1}(S(T^{n-1}(x)))$

Thus, the induction hypothesis being true, we can assert for all n:

$$Rn(x) = S(x)+Q(S(T(x)))+Q^2(S(T^2(x)))+...+Q^{n-1}(S(T^{n-1}(x)))$$

Assuming that $T^n(x)$ is different of $T^k(x)$ for all n and k with n different of k (i.e that T is acyclic), this formula can be transform in an iterative program to compute R(x) which is the limit of Rn(x) when n grows. This program is given below :

```
Procedure evaluate(R,x);
```

```
i,n : integer;
  Begin
    n := 0;
    R := S(x);
    DT := T(x);
    while DT <> 0 do
      n := n+1;
      DR := S(DT);
      FOR i:=1 TO n do DR := Q(DR) od;
      R:= R + DR;
      DT := T(DT);
    od;
  end;
```

The above procedure is correct for any recursive predicate R defines by linear clauses (i.e., clauses which leads to linear series). Let us point out once more that x can be any set of values. The addition is the set union and the function composition is the one defined above.

The previous result can be applied to the family friend example. The serie obtained was a linear one as defined above where:

Rn is called FFn;
S is called F;
Q is the identity;
T is called P which is acyclic.

Therefore, the procedure to compute FF(x) is :

```
Procedure evaluate(FF,x);
i,n : integer;
  Begin
    n := 0;
    FF := F(x);
    DP := P(x);
    while DP <> 0  do
      n := n+1;
      DFF := F(DP);
      FOR i:=1 TO n do DFF := DFF od;
      FF := FF + DFF;
      DP := P(DP);
    od;
  end;
```

As Q is the identity function the loop to built Q is obviously useless. Therefore, we get the simplified procedure to retrieve the family friend of x :

```
Procedure evaluate(FF,x);
  Begin
    FF := F(x);
    DP := P(x);
    while DP <> 0 do
      DFF := F(DP);
      FF := FF + DFF;
```

```
    DP := P(DP);
  od;
end;
```

Also, as P(x) is indeed P:1->2(x) and F(x) is indeed F:1->2(x), we can move back to relational algebra using SQL notations. Thus, we get the following procedure to evaluate the family friend of x (where x may be toto or any set of people) :

```
Procedure evaluate(FF,x);
  Begin
    FF := SELECT PERSON2 FROM FRIEND
                 WHERE PERSON1 = x;
    DP := SELECT PARENT FROM PARENT
                 WHERE CHILDREN = x;
    while DP <> 0 do
      DFF := SELECT PERSON2 FROM FRIEND
                    WHERE PERSON1 IN DP;
      FF := FF UNION DFF;
      DP := SELECT PARENT FROM PARENT
                   WHERE CHILDREN IN DP;
    od;
  end;
```

The previous result can also be applied to the same generation cousins example. The serie obtained was a linear one as defined above where:

Rn is called SGn;
S is the identity;
Q is called P';
T is called P which is acyclic.

Therefore, the procedure to compute SG(x) is:

```
Procedure evaluate(SG,x);
i,n : integer;
  Begin
    n := 0;
    SG := x;
    DP := P(x);
    while DP <> 0  do
      n := n+1;
      DSG := DP;
      FOR i:=1 TO n do DSG := P'(DSG) od;
      SG := SG + DSG;
      DP := P(DP);
    od;
  end;
```

As P(x) is indeed P:1->2(x) and P'(x) is indeed P:2->1(x), we can move back to relational algebra using SQL notations. Thus, we get the following procedure to evaluate the same generation cousins of x (where x may be Agamemnon or any set of people) :

```
Procedure evaluate(SG,x);
i,n : integer;
  Begin
    n := 0;
    SG := x;
    DP := SELECT PARENT FROM PARENT
                WHERE CHILDREN = x;
    while DP <> 0  do
      n := n+1;
      DSG := DP;
      FOR i:=1 TO n do
        DSG := SELECT CHILDREN FROM PARENT
                    WHERE PARENT IN DSG;
      od;
      SG:= SG UNION DSG;
      DP := SELECT PARENT FROM PARENT
                  WHERE CHILDREN IN DP;
    od;
  end;
```

5.3 Bilinear and Polynomial systems

Definition 7: Bilinear system

> A bilinear system is a set of recursive rules which leads to a bilinear serie of the form :
>
> R0(x) = 0
>
> Rn(x) = S(x) + Rn-1(Rn-1(x))
>
> where S is a function derived from base relations or from non recursive expressions of base relations.

Bilinear series does not include but are more complex than linear series. The even ancestor example is an example of bilinear series. Such series may be solved. Indeed, the solution is:

$$Rn(x) = S(x) + S^2(x) + \ldots + S^{(n-1)**2}(x).$$

This formula can be applied to generate a program, for example to compute the ancestor of even generation.

In a much more general way, we can introduce polynomial systems as follows.

Definition 8: Polynomial system

> A polynomial system is a set of recursive rules which leads to a polynomial serie of the form :
>
> R0(x) = 0
>
> Rn(x) = S(x) + F1(Rn-1)(x) + F2(Rn-1)(x) + ...
>
> where F1, F2 ... is some composition of the argument relation with some base or base derived relations.

Such rules which are a generalization of the linear one are too complex to be solved in our current knowledge.

6. CONCLUSION

In this paper, we have presented a framework for compiling a large class of recursive queries into SQL optimized programs. The concerned class includes all query-rules which may be transformed into functional expressions, that is what we call the tree query-rule class. The approach may also be used to isolate non classical relational operators as transitive closures. It is also important to point out that the approach allows us to classify query-rules into regular, linear and polynomial systems.

We claim that the approach is easy to implement for at least linear query-rule systems. One way of doing is to keep the general SQL program which solves such systems and to instantiate the parameters using an analysis of the query and the rules. This analysis would build the query-rule connection graph, derive the recurrent equation and verify that it is a linear one. Then, parameter instantiation could be done. One problem, which appears in the same generation cousin example, is a possible simplification of the program. This is probably feasible with an algorithm.

Another possible implementation which is under investigation in our project is to implement fixpoint operators as transitive closure and external closure. External closure is an operator which closes a relation T (after a possible restriction) at level n, then join the result with a relation S, then performs n join with a relation Q, finally project on S schema and collect the results for all n, with S. Such an operator is exactly what is needed to compute the answer of linear systems. We could also distinguish left external closure where a relation S is composed at left by Q, and right external closure where it is composed at right by T. There is much more work that can be done to generalize such operators with complex conditions (with < or > comparators) or with arithmetic computations at each step. We believe that the efficient implementation of such operators is necessary to process efficiently recursive queries. Such queries are frequent in real applications as shown by the study of one application at the French RATP (i.e. the enterprise which manages the metro in PARIS).

Acknowledgments

We gratefully acknowledge the SABRE team for helpfull discussions and more specifically Eric SIMON for its incisive comments of an earlier version of this paper. We would like also to thank Francois Bancilhon for sending us some very nice papers.

REFERENCES

[Aho79] Aho A.V., Ullman J.D.
"Universality of Data Retrieval Languages"
Conf. on POPL, San-Antonio, Texas, 1979.

[Bancilhon85a] Bancilhon F.
"Evaluation des Clauses de Horn dans les Bases de

Donnees Relationnelles"
To appear, PRC BD3 Book, in French, Eyrolles 86

[Bancilhon85b] Bancilhon F.
"Naive Evaluation of Recursively Defined Predicates"
MCC Internal Report, 1985

[Bancilhon85c] Bancilhon F., Maier D., Sagiv Y., Ullman J.D.
"Magic Sets and Other Strange Ways to Implement Logic Programs"
MCC Technical Report, 1985

[Bernstein79] Bernstein P.A., Chiu D.W.
"Using Semi-joins to Solve Relational Queries"
Technical Report CCA 01-79, Jan. 1979
Also in JACM, V28, N1, Jan. 1981, Pp 25-40.

[Chandra82] Chandra K.A., Harel D.
"Horn Clauses and the Fixpoint Query Hierarchy"
Proc. ACM Symp. on Principles of Databases , 1982

[Chandra85] Chandra K.A., Harel D.
"Horn Clauses Queries and Generalizations"
The Journal of Logic Programming, N1, 1985

[Chang81] Chang C.
"On Evaluation of Queries Containing Derived Relations in a relational database"
In [Gallaire81].

[Gallaire81] Gallaire H., Minker J., Nicolas J.M.
"Advances in Database Theory"
Books, Vol 1, Plenum Press, 1981

[Gallaire84] Gallaire H., Minker J., Nicolas J.M.
"Logic and Databases : A Deductive Approach"
ACM Computing Surveys, Vol 16, No 2, June 1984.

[Henschen84] Henschen L.J., Naqvi S.A.
"On Compiling Queries in Recursive First-Order Databases"
JACM Vol 31, No 1, Jan 1984

[Lozinskii85] Lozinskii E.L.
"Evaluating Queries in Deductive Databases by Generating Subqueries"
IJCAI Proc., Pp 173-177, Los Angeles. August 1985.

[Tarski55] Tarski A.
"A Lattice Theoretical Fixpoint Theorem and its Application"
Pacific Journal of Mathematics N5, Pp 285-309, 1955

[Ullman85] Ullman J.D.
"Implementation of Logical Query Languages for Databases"
ACM SIGMOD, Austin, Texas 1985.

[Vieille85] Vieille L.
"Recursive Axioms in Deductive Databases : The Query-Subquery Approach"
ECRC Internal Report 1985.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique